Random Number Generators for Supercomputers

Richard P. Brent

Computer Sciences Laboratory

Australian National University

Outline

- Requirements for RNGs
- Linear Congruential RNGs
- Generalized Fibonacci RNGs
- Comments on some available RNGs
- Vectorization
- Initialization
- Implementation on the Fujitsu VP2200
- Conclusions

Introduction

Pseudo-random number generators are widely used in simulation.

A program running on a supercomputer might use 10^8 random numbers per second for many hours.
Small correlations or other deficiencies could easily lead to spurious results.

We need to have confidence in the statistical properties of random number generators.

Some Requirements

- Uniformity This is the easiest requirement to achieve, at least when considered over the whole period.

- Independence d-tuples should be uniformly and independently distributed in d-dimensional space ($d \leq 6$). Subsequences (e.g. odd/even) of the main sequence should be independent.

• Long Period The period (length of a cycle in the random numbers generated) should be large, certainly at least 10^{12} and preferably much larger.

A library generator[1] on the Fujitsu VP2200/10 has period 2^{31} and runs through a complete cycle in less than one minute. This and similar generators are obsolete and should be avoided.

[1]RANU2 in the SSL II library.

• Repeatability For testing and development it is useful to be able to repeat a run with exactly the same sequence as was used in another run, but not necessarily starting from the beginning of the sequence.

Thus, it should be easy to save all the "state" information required to restart the generator.

• Portability For testing and development it is useful to be able to generate exactly the same sequences on different machines (possibly with different wordlengths).

At the least, a machine with a long wordlength should be able to simulate a machine with a shorter wordlength, without much loss of efficiency.

• Disjoint Subsequences If a simulation is run on a parallel machine or on several independent machines, the sequences of random numbers generated on each machine must be independent.

• Efficiency Only a few arithmetic operations should be required to generate each random number.
Subroutine call overheads should be minimised.
The implementation should exploit any vector or parallel capabilities of the computer.

Linear Congruential RNGs

Introduced by D. H. Lehmer in 1948.

$$U_{n+1} = (aU_n + c) \bmod m$$

where $m > 0$ is the modulus, a is the multiplier, and c is an additive constant.

Often $m = 2^w$ is chosen as a convenient power of 2. In this case it is possible to get period m. However, $w \leq 32$ is not large enough ($2^{32} << 10^{12}$).

Generalized Fibonacci Generators

$$U_n = U_{n-r} \, \theta \, U_{n-s}$$

where r and s are fixed "lags" and θ is some binary operator, e.g. addition (mod m).

$r = 2, s = 1, \theta = +$ gives the Fibonacci sequence (mod m). This is certainly not a satisfactory random number generator, because only 4 of the 6 possible orderings of 3 numbers actually occurs. We need larger r and s.

Some Available RNGs

Many implementations of linear congruential generators are available. They usually have a period which is too short and do not give good d-dimensional uniformity for $d > 3$ (Marsaglia[2]).

Marsaglia dislikes Tausworthe (shift register, $\theta = \oplus$) RNGs because they fail the "birthday spacings" test.

[2]"Random numbers fall mainly on the planes"

Available RNGs cont.

Marsaglia now recommends "Very Long Period" generators, but these also fail the birthday spacings test unless care is taken[3]. They also more difficult to vectorise than linear congruential or generalised Fibonacci generators.

[3]Perhaps this is why Marsaglia combines his VLP generator with a different generator.

Vectorization

It is easy to vectorise both linear congruential and generalised Fibonacci RNGs. This is only useful if batches of random numbers are generated together. Thus, the interface to a library routine should allow an array of random numbers to be returned[4].

[4] This applies even on a workstation, because of the reduction in subroutine-call overheads.

Initialization

Using the theory of generating functions, it is possible to "skip ahead" *n* terms in the sequence for a generalized Fibonacci RNG in *$O(\log n)$* arithmetic operations[5]. The idea is similar to that of forming *n*-th powers by squaring and multiplication.

[5] In practice it is sufficient (and faster) to apply this technique just to the least significant bits.

Initialization cont.

This technique allows us to guarantee that different seeds give different sequences for all practical purposes (e.g. use segments of the full sequence separated by more than 10^{18} numbers).

This facility is useful for performing independent simulations on a serial computer, or on each processor of a parallel computer.

Implementation on the Fujitsu VP2200/10

A class of generalized Fibonacci RNGs has been implemented on ANU’s VP2200. Statistical properties are good. The lags *r* and *s* are automatically selected, depending on the size of workspace provided by the user. The implementation uses floating-point arithmetic (56-bit fraction).

The period is between 10^{54} and 10^{13411} (depending on *r*).

Speed of our Implementation

Provided batches of several hundred random numbers are generated at once (so vectorization is effective) each random number requires about 2.21 machine cycles[6].

Thus, we can generate more than *140 million* uniformly distributed random numbers per second.

[6]Here a machine cycle is 3.2 nanoseconds.

Conclusions

- **The class of generalized Fibonacci RNGs is attractive for vector and parallel computers because of the potential for speed, long period, and good statistical properties.**

- **Our implementation on the Fujitsu VP2200/10 at ANU satisfies the requirements (uniformity, independence, repeatability, ...) mentioned earlier for a good library RNG.**